Étape 7 : fourmis et fourmilières, partie 1.

Jamila Sam & Jean-Cédric Chappelier, 2018

Version : 1.0

But

Modéliser les fourmis et les fourmilières.

Description

Nous allons poursuivre la modélisation des fourmis et des fourmilières précédemment ébauchées. Nous terminerons cette conception lors de la prochaine étape.

Il s'agit pour l'essentiel d'étoffer ces composants du programme en établissant notamment le lien entre fourmis et fourmilières. Les fourmilières et les fourmis sont appelées à interagir avec l'environnement (les premières pour faire émerger des fourmis qui s'y déplaceront et les secondes pour retrouver leur fourmilière par exemple). Pour que ces composants ne puissent agir sur l'environnement autrement que ce qu'ils sont censés faire, nous recourrons à l'utilisation de nouvelles «vues sur l'environnement», selon le schéma que nous avons pris l'habitude d'utiliser.

La figure 1 donne une vue d'ensemble de l'architecture à laquelle vous allez aboutir au terme de cette étape et de la suivante.

Implémentation

Ouvrières et soldates

Les fourmis, qu'elles soient ouvrières ou soldates appartiennent systématiquement à une fourmilière. Pour établir ce lien, introduisez un attribut identifiant la fourmilière à laquelle appartient la fourmi. Cet identifiant sera de type `Uid` (pour « Unique identifier ») qui vous est fourni. Ajoutez de plus une méthode `getAnthillId()` retournant cet identifiant.

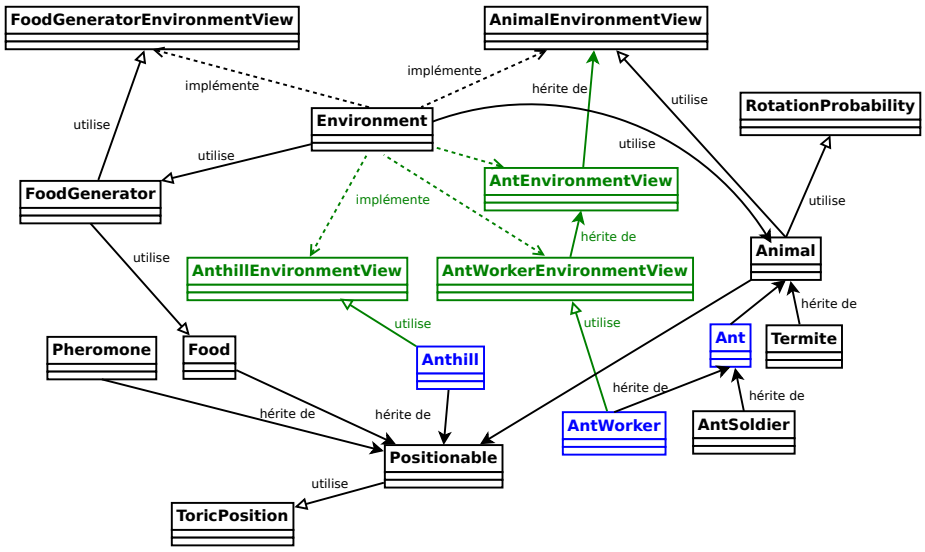

Figure 1: Architecture codée au terme des étapes 7 et 8 (en noir les composants déjà codés, en vert les nouveaux composants, en bleu ceux retouchés)

Par ailleurs, nous commençons à nous intéresser aux spécificités des fourmis. Les ouvrières ont pour rôle de transporter de la nourriture. Vous introduirez donc comme attribut de la classe `AntWorker` la quantité de nourriture qu'elle transporte, de type `double` et initialisé à zéro. Modifiez également la méthode `getFoodQuantity()` afin qu'elle retourne cette valeur.

Les constructeurs initialiseront la position de la fourmi et l'identifiant de la fourmilière à laquelle elle appartient au moyen de valeurs passées en paramètres.

En particulier, le constructeur de la classe `Ant` sera adapté se sorte à prendre en paramètres la position de la fourmi, ses points de vie, sa durée de vie initiale et sa fourmilière.

Enfin, de façon analogue à ce que vous avez fait pour les termites,

- les constructeurs de `AntWorker` et `AntSoldier` initialiseront :
  - à `ANT_WORKER_HP` les points de vie des fourmis ouvrières et à `ANT_WORKER_LIFESPAN` leur durée de vie ;
  - à `ANT_SOLDIER_HP` les points de vie des fourmis soldats et à `AN_SOLDIER_LIFESPAN` leur durée de vie ;
- les fourmis ouvrières et soldates auront des valeurs spécifiques pour la vitesse, `ANT_WORKER_SPEED` et `ANT_SOLDIER_SPEED` respectivement.

Vous doterez enfin les ouvrières d'une méthode `toString()` retournant leur représentation textuelle selon le format de l'exemple suivant :

```
Position : 5.0, 10.0
Speed : 100.0
HitPoints : 500
LifeSpan : 50.00
Quantity : 2.35
```

La dernière ligne donne la quantité de nourritures transportée par la fourmi.

Les libellés `Position`, `Speed`, `HitPoints`, `LifeSpan` et `Quantity` sont importants pour une bonne intégration avec l'interface graphique. Vous veillerez à réutiliser proprement les méthodes `toString()` héritées de plus haut.

Fourmilières

Les fourmilières (classe `Anthill`), qui serviront de point de ralliement aux fourmis, seront caractérisées essentiellement par :

- le stock nourriture qu'elles contiennent (la nourriture stockée assurant la pérennité de la fourmilière). Le stock vaudra zéro au début et aura le type `double` ;
- leur identifiant unique de type `Uid` ;
- la probabilité de voir sortir une fourmi ouvrière, un `double`.

Et, bien sûr, une fourmilière doit être positionnable dans notre monde torique.

Une fourmilière sera construite au moyen d'une position (torique) et d'une probabilité passées en paramètre. La valeur par défaut vaudra `ANTHILL_WORKER_PROB_DEFAULT`. L'identifiant unique associé à la fourmilière sera initialisé, au moment de la construction, au moyen de la méthode statique `createUid()` de la classe fournie `Uid`.

Une fourmilière sera capable de recevoir une quantité donnée de nourriture (rapportée par une fourmi). Pour cela, définissez la méthode

```
void dropFood(double toDrop)
```

dans la classe `Anthill`. La quantité reçue sera simplement ajoutée au stock.

Définissez également les méthodes `getFoodQuantity()` et `getAnthillId()`, retournant les attributs correspondants.

Vous doterez enfin les fourmilières d'une méthode `toString()` retournant leur représentation textuelle selon le format de l'exemple suivant :

```
Position : 10.5, 20.0
Quantity : 50.5
```

Les libellés `Position` et `Quantity` sont importants pour une bonne intégration avec l'interface graphique. Vous veillerez à réutiliser proprement les méthodes `toString()` héritées de plus haut.

Extension de la classe `Environment`

Ajout de fourmilières à l'environnement

L'environnement doit désormais pouvoir aussi contenir un ensemble de fourmilières. Utilisez pour cela une `List` Java.

Programmez ensuite la méthode publique

```
void addAnthill(Anthill anthill)
```

afin de permettre l'ajout d'une fourmilière donnée à l'environnement.

Nous souhaiterions que les fourmilières puissent ajouter des fourmis (et uniquement des fourmis, pas des termites) à l'environnement. Nous aurons donc besoin d'utiliser des vues sur l'environnement comme nous l'avons fait pour `FoodGenerator`. Créez l'interface `AnthillEnvironmentView` qui comportera comme méthode

```
void addAnt(Ant ant)
```

ajoutant la fourmi passée en paramètre à l'environnement (et lançant une `IllegalArgumentException` si `ant` vaut `null`).

Faites en sorte que la classe `Environment` implémente cette nouvelle interface et définissez la méthode en question simplement comme un appel à `addAnimal()` (l'environnement peut, lui, ajouter n'importe quel animal et à ce stade (dans la méthode `addAnt()`), il « sait » que c'est une fourmi).

La vue des fourmis sur l'environnement

Une fourmi (quelconque) aura besoin d'interagir avec sa fourmilière. Les fourmis ouvrières devront de plus avoir la possibilité d'aller chercher de la nourriture. Nous créerons donc une vue commune à toutes les fourmis qui permettra d'obtenir une référence vers leur fourmilière. La vue spécifique aux fourmis ouvrières aura une méthode supplémentaire permettant d'obtenir des informations sur la nourriture aux alentours.

Commencez par créer l'interface `AntEnvironmentView` qui héritera de l'interface `AnimalEnvironmentView`. Il sera vide pour le moment.

Créez ensuite l'interface `AntWorkerEnvironmentView` qui héritera de l'interface précédemment définie `AntEnvironmentView`. Il vous est demandé d'ajouter le prototype des méthodes suivantes :

- `Food getClosestFoodForAnt(AntWorker antWorker)` retournant une référence sur la source de nourriture la plus proche *perceptible* par une fourmi ouvrière `antWorker` ; la valeur `null` sera retournée si aucune source de nourriture n'est détectée ;

- `boolean dropFood(AntWorker antWorker)` retournant `true` si `antworker` peut ajouter la quantité de nourriture qu'elle transporte à sa fourmilière. Ce sera le cas uniquement si sa fourmilière est *perceptible*.

Une fourmi ne peut percevoir que ce qui est dans son rayon de perception. Ce dernier est donné (pour toutes les fourmis) par `ANT_MAX_PERCEPTION_DISTANCE` (à récupérer via `getConfig()` comme d'habitude). Si la distance entre la fourmi et l'entité recherchée est *strictement supérieure* à ce rayon, l'entité en question n'est pas perceptible.

Note : `dropFood` ne se préoccupe pas de la quantité effective de nourriture transportée. Elle peut donc retourner `true` même si la quantité transportée est nulle.

Il reste enfin à faire implémenter les interfaces `AntEnvironmentView` et `AntWorkerEnvironmentView` à la classe `Environment`. Vous pouvez pour cela vous aider de la méthode statique `closestFromPoint()` de `Utils` qui prend comme paramètres une entité `Positionable` et une collection d'entités qui sont également `Positionable`.

Tests et soumission

Tests locaux

Nous vous encourageons, comme toujours à continuer d'ajouter vos propres tests ponctuels dans le fichier `ch/epfl/moocprog/tests/Main.java`

Voici une exemple de tests simples que vous pouvez effectuer :

```
// Quelques tests pour l'étape 7
  Anthill anthill = new Anthill(new ToricPosition(10, 20));
  System.out.println("Displaying an anthill");
  System.out.println(anthill);
  env = new Environment();
  env.addAnthill(anthill);
  Food f3 = new Food(new ToricPosition(15, 15), 20.);
  Food f4 = new Food(new ToricPosition(40, 40), 15.);
  env.addFood(f3);
  env.addFood(f4);
  System.out.println();

  AntWorker worker = new AntWorker(new ToricPosition(5, 10), anthill.getAnthillId());
  System.out.println("Displaying a worker ant");
  System.out.println(worker +"\n" );

  System.out.print("Can the worker ant drop some food in its anthill : ");
```

```java
// true car la fourmi est assez proche de sa fourmilière
System.out.println(env.dropFood(worker));
System.out.println("Displaying the anthill after the antworker dropped food:");
// aucun changement car la fourmi ne transporte pas de nourriture
System.out.println(anthill);

System.out.println("\nClosest food seen by the worker ant:" );
// la fourmi ne « voit » que f3
// si l'on n'avait que f4, l'appel suivant retournerait null
System.out.println(env.getClosestFoodForAnt(worker));
```

Ce qui devrait résulter dans l'affichage suivant :

```
Displaying an anthill
Position : 10.0, 20.0
Quantity : 0.00

Displaying a worker ant
Position : 5.0, 10.0
Speed : 100.0
HitPoints : 500
LifeSpan : 50.000000
Quantity : 0.00

Can the worker ant drop some food in its anthill : true
Displaying the anthill after the antworker dropped food:
Position : 10.0, 20.0
Quantity : 0.00

Closest food seen by the worker ant:
Position : 15.0, 15.0
Quantity : 20.00
```

Libre à vous d'enrichir et d'étoffer ces tests selon ce qui vous semble pertinent.

Continuez bien sûr à utiliser l'interface graphique pour tester globalement le comportement de tout le système. A ce stade vous devriez obtenir une exécution analogue à celle de l'étape précédente, mais avec l'affichage de fourmilière en plus. Utilisez le boutons `Quantity` pour vérifier que les fourmilières ont bien des stocks de nourriture vide au départ, selon l'exemple de la figure 2.

Soumission

Pour soumettre votre devoir au correcteur automatique, il faut créer un fichier ZIP contenant **tous** vos fichiers sources personnels, depuis la racine `ch/` ; ceux de cette étape, mais aussi ceux de l'étape précédente. Concrètement, pour ce devoir ci, votre fichier ZIP doit donc contenir exactement les fichiers suivants :

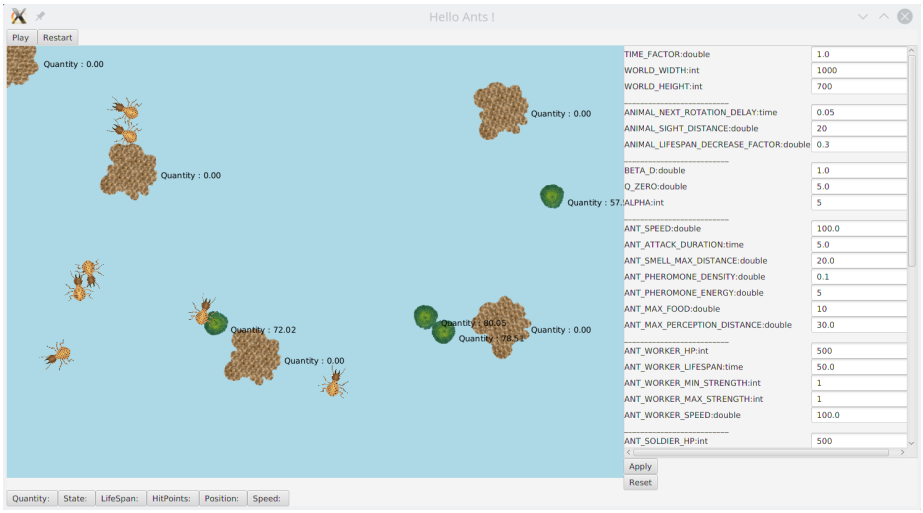

Figure 2: Exemple d'exécution après introduction des fourmilières

`ch/epfl/moocprog/AnimalEnvironmentView.java`
`ch/epfl/moocprog/Animal.java`
`ch/epfl/moocprog/AntEnvironmentView.java`
`ch/epfl/moocprog/AnthillEnvironmentView.java`
`ch/epfl/moocprog/Anthill.java`
`ch/epfl/moocprog/Ant.java`
`ch/epfl/moocprog/AntSoldier.java`
`ch/epfl/moocprog/AntWorkerEnvironmentView.java`
`ch/epfl/moocprog/AntWorker.java`
`ch/epfl/moocprog/Environment.java`
`ch/epfl/moocprog/FoodGeneratorEnvironmentView.java`
`ch/epfl/moocprog/FoodGenerator.java`
`ch/epfl/moocprog/Food.java`
`ch/epfl/moocprog/Pheromone.java`
`ch/epfl/moocprog/Positionable.java`
`ch/epfl/moocprog/RotationProbability.java`
`ch/epfl/moocprog/Termite.java`
`ch/epfl/moocprog/ToricPosition.java`

Note : si c'est plus pratique pour vous, vous *pouvez* aussi faire un zip qui contient tout le sous-répertoire `ch/epfl/moocprog`, y compris, donc, les fichiers que nous vous avons fournis. Ces fichiers seront simplement ignorés par le correcteur automatique (et remplacés par les nôtres). L'énoncé « *Procédure de soumission* » de la semaine 1 vous indique comment procéder. **Avant de passer à l'étape suivante, n'oubliez pas de faire une sauvegarde de l'étape en cours**, comme indiqué dans le même document.